AIML402: Assignment 1 report

Junyi Shen (8386129)
August 12, 2024

The assignment is to develop an agent to play the game of Raj and compete with other agents, such as random_agent.py, value_agent.py, and valueplus_agent.py. In this report, I will introduce the strategies I use, explain how the algorithm works, and illustrate the performance of my agent under different test cases.

1 Strategy

The approach to developing the agent is depth-limited minimax algorithm, with alpha-beta pruning to improve the performance. A heuristic evaluation function is elaborately designed to improve the competitiveness of the agent.

1.1 Why choose minimax algorithm

The Raj game is not an alternating game. In each round, players show their cards at the same time instead of showing them one player after another. However, minimax algorithm is still a good way to develop the agent. We can assume that in each round, my agent shows the bidding before the opponent, allowing the opponent to make a decision after seeing the card my agent uses. This is the worst case for my agent and it will enable it to react wisely even in a bad situation.

1.2 Algorithm introduction

In my minimax algorithm, my agent acts as the MAX and the opponent is the MIN. In MAX's move, MAX only takes a card out from the cards in hand and there is no comparison. A round only ends after the MIN's move, in which the opponent selects a card and compares it with the card MAX just used. After comparing the cards and updating the banks of both players, the round ends and it is the MAX's move again.

My minimax algorithm is implemented by recursion to depth-first search a tree. Initially, the root state is created with the current percepts. A root node with the root state is then created and passed into the get_best_child() recursive function, whose goal is to find the best child node in order to get the best action. In this function, the next possible states are found and iterated. The evaluation value is calculated only if the depth limit is reached or the state is already a terminal state, otherwise, the child node is processed and this function is called again recursively. After processing a child node(calculate evaluation value or recursively call its children), back up its evaluation value to the parent. Eventually, the get_best_child() function will return the best child of the root node.

1.3 Limited search depth

A limited depth of 4 is applied to the search tree for two reasons. First, if the search is too deep, it requires a long time to process too many nodes, which affects the efficiency

of the algorithm. Second, since the order of the items to bid on is chosen randomly, we don't know what the next items to bid on are, so we can only assume choosing items in order, which affects the effectiveness of a deep-depth search(Tests show that the improvement of scores is very limited when I increase the depth to over 4). A depth limit of 4(two rounds) is appropriate. It only requires a limited search time but has good competitiveness.

1.4 Alpha-beta pruning

To improve the performance of my algorithm, alpha-beta pruning is applied. In the get_best_child() recursive function, *alpha* and *beta* are passed as parameters with initial values of negative infinite and positive infinite, respectively. *Alpha* or *beta* is assigned a new value when a better child assigns its evaluation value to the parent. When $alpha \geq beta$, the subsequent sibling nodes can be pruned because in this situation, the best value MAX or MIN(depends on the level) can guarantee is better than the best value in this branch. Thus, there can be a better choice for MAX or MIN and it is not necessary to process the sibling nodes.

1.5 Heuristic evaluation function

The heuristic evaluation function is used for calculating the utility of each leaf node. It is very important because it determines how good a state is and it directly affects the behavior of the agent.

In my evaluation function, the final score contains two parts. The first part is to calculate the potential score, which indicates the potential value to win in future rounds. The potential value's calculation is as follows:

$$potential\_value = items\_left\_abs\_average \times card\_value\_compare\_score$$

In this formula, *items_left_abs_average* is the average absolute value of the items left, which indicates the average value to bid on later on in the game. To calculate the *card_value_compare_score*, sort the cards in hand of both players, then compare each position of the two players' cards. For each position, if my card's value is greater than the opponent's card value, score plus 1. If the value is less than the opponent's, score minus 1. This *card_value_compare_score* indicates the competitiveness of cards in hand for the subsequent rounds. By multiplying *items_left_abs_average* and *card_value_compare_score*, we can estimate the potential value my agent can get in the rest of the game.

The second part of the evaluation function is the difference between my bank and the opponent's bank, which is calculated by subtracting opponent's bank from my bank. This is the score I have already got over the opponent's.

Finally, add the two parts together and get the final score:

$$total\_score = potential\_value + bank\_diff \times 1.2$$

I applied a 1.2 multiplier to the bank difference because the value already in bank, which is confirmed, is slightly more important than the potential value to get in the future. Thus, the bank difference should contribute a little bit more than the potential value.

Opponent	Number of Games	Random seed	Total running time(in seconds)	My agent average score	Opponent average score	My score/ Opponent score
random_agent.py	1000	0	8	4.93	1.86	2.65
random_agent.py	1000	1	8	4.98	1.90	2.62
random_agent.py	1000	2	8	4.99	1.80	2.77
value_agent.py	1000	0	8	3.67	2.36	1.56
value_agent.py	1000	1	8	3.78	2.42	1.56
value_agent.py	1000	2	8	3.74	2.40	1.56
valueplus_agent.py	1000	0	8	3.58	2.21	1.62
valueplus_agent.py	1000	1	8	3.65	2.27	1.61
valueplus_agent.py	1000	2	8	3.67	2.25	1.63

Table 1: My agent's performance. Test environment: MacBook Pro 2019

2 Agent performance

2.1 Test cases

To test the performance of the agent, I use it to compete with three different other agents:

1. random_agent.py. It simply uses a random card in hand in each round.
2. value_agent.py. It always uses the card with the closest value to the absolute value of the item.
3. valueplust_agent.py. It chooses the card with one higher value than the closest-valued one if possible.

For each opponent, I applied three tests with a random seed of 0, 1, and 2. In each test, I run 1000 games to evaluate the running time and compare the score of my agent and the opponent.

The testing environment is MacBook Pro 2019, with a processor of 2.6 GHz 6-Core Intel Core i7.

2.2 Test result

As Table 1 shows, we can separate the result into two parts.

1. Efficiency

   For each test, the running time of my agent on the test machine(MacBook Pro 2019) is approximately 8 seconds for 1000 games. The fast speed of the algorithm is due to two reasons. First, the depth limit is only 4, or two rounds. The search tree doesn't process many nodes even in the initial states. Second, alpha-beta pruning allows the program to eliminate many nodes that are not necessary to process, which contributes to the efficiency of the algorithm(I have tested that it takes about 23 seconds to run 1000 games if I don't adopt the alpha-beta pruning strategy).

2. Competitiveness

   I calculate the competitiveness value of the agent by dividing my agent's average score by the opponent's average score in each test. This value indicates for the 1000 games, how many times my agent's score is than the opponent's. The result shows that when competing with random_agent.py, my agent can get approximately 2.7 times of scores than the opponent. While competing with value_agent.py or valueplus_agent.py, the number is around 1.6, which is smaller because these two agents adopt a wiser strategy than the random agent. My agent can defeat the three agents in each 1000-game test.

3 Conclusion

In this assignment, I developed an agent that can play well in the Raj game in terms of both efficiency and competitiveness. By assuming the opponent bids after me in each round, I adopted the minimax algorithm to develop this agent. Additionally, limited depth and alpha-beta pruning contribute to the efficiency of the programme, allowing the agent to play a large number of games in a short time.

To enhance the competitiveness of my agent, I improved the heuristic evaluation function many times. Initially, I simply calculated the difference between the banks of two players, then additionally considered the sum of cards in their hands and calculated their difference. However, these strategies seem not to be competitive enough. Eventually, the evaluation function was modified to consider not only the banks but also the potential value the agent can get in the future. To calculate the potential value, the cards in hand in each position of both players are compared independently, and the value of items to bid is also considered.

The assignment deepens my understanding of adversarial search especially depth-limited minimax algorithm and alpha-beta pruning. It also encourages me to develop and keep improving the heuristic evaluation function, which is crucial for search problems.